UNIVERSITY OF OSLO
Department of Informatics

Coupling Diffpack with Sparskit and pARMS

Master's thesis

Anders Schau Knatten

October 31, 2007

Preface

Partial differential equations (PDEs) can be used to describe a wide range of physical phenomena, from modelling tsunamis, fluid flow and heat, to quantum mechanics, weather forecast and electrical activity in the heart. These problems are usually very complex, and require huge amounts of calculations. Thanks to modern computers in general, but parallel computers especially, it is possible to solve bigger and bigger problems.

Over the years, several general software packages have been developed that can be used in the different stages of solving partial differential equations. The aim of this thesis is to extend Diffpack, a general tool for solving PDEs, with pARMS, a tool for solving systems of linear equations on parallel computers. As part of the process, Sparskit, a similar tool for serial computers, is also integrated with Diffpack.

Goals There are two goals with this thesis.

1. Diffpack already has a toolbox for solving linear systems in parallel. Is it possible to extend this toolbox in a flexible way, allowing the Diffpack user to use techniques from pARMS as though they were just another part of Diffpack?

2. What can be gained from this extension? Can the introduction of pARMS make Diffpack simulators more efficient?

Overview of the Text The main body of the text is concerned with the integration of Diffpack with Sparskit and pARMS, and benchmarking the pARMS integration. First, Chapter 1 gives a quick background on grids, partitioning and overlap. Chapter 2 continues with a background on solution of linear systems on serial and parallel computers. Then chapter 3 introduces Diffpack, Sparskit and pARMS. Chapter 4 treats integration of Fortran and C++, which is necessary for coupling Diffpack with Sparskit. Then, Chapter 5 presents the actual coupling of the three software packages. The performance of the coupling is then discussed with two application examples in Chapter 6 . Finally, Chapter 7 concludes the thesis and suggests some further work.

Appendix A is a reference for the matrix storage format "Compressed Row Storage (CRS)", used extensively in all three software packages. Appendices B and C describes installation of Sparskit and pARMS, while D describes how to use pARMS with Diffpack.

Acknowledgements First, I would like to thank my supervisor, prof. Xing Cai, for his help and encouragement, and his always swift replies to my questions. I would also like to thank Simula Research Laboratory in general, for providing me with good working conditions, and for always including the students in whatever seminars and events were taking place. Thanks also to my fellow students, especially Mads Fredrik Skoge Hoel, Guo Wei Ma and Martin Burheim Tingstad, for interesting discussions and great company.

Next, I would like to thank my fiancée Charlotte Kristensen and my family for always being supportive, and for understanding my occasional mental absence.

Finally I would like to thank Marius Kristiansen Kamnes and Asbjørn Aage Ulsberg for initiating a series of events leading me into the field of scientific computing.

Contents

1 Grids and partitioning

1.1 Grids

Finite element solutions of PDEs are defined on a grid. Representation of grids plays a big part in the integration with pARMS, and this chapter provides a background for understanding the necessary concepts and terms used.

There are various methods for solving PDEs, this thesis focuses on the finite element method (FEM). In FEM, the continuous problem domain is divided into a discrete grid, which is built up by elements. Each element is defined by a number of nodes, see Figure 1. As shown in the figure, grids can be structured or unstructured. A common choice of element shapes for the finite element method is triangles, but other shapes, like rectangles, can also be used. All these examples are two dimensional, but they generalize to one dimension (elements are line segments) and higher dimensions (tetrahedra and hexahedra in three dimensions), see Figure 2.

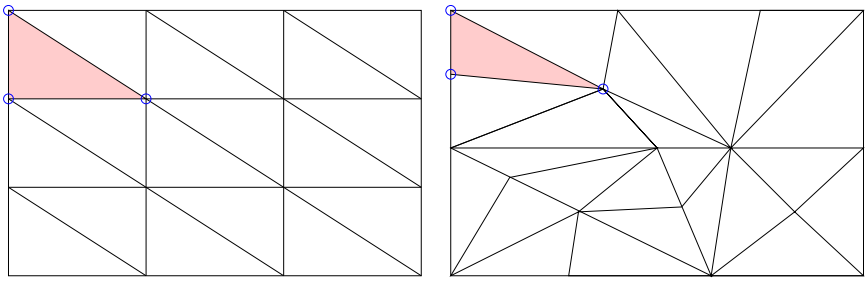

Figure 1: Left: A structured grid. Right: An unstructured grid. Examples of elements are marked in red, and their corresponding nodes with a a blue circle.

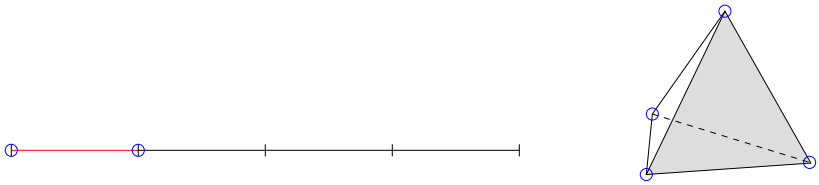

Figure 2: Left: A one dimensional grid. One element in the grid is marked in red, and the nodes belonging to that element are marked in blue. Right: A three dimensional grid element.

1.2 Partitioning

When solving a problem in parallel the problem needs to be divided in some way between the processes. One way to do this is to construct the linear system as usual, and then divide this system between the processes. A better way is to partition the grid itself, and then let each process construct only the parts of the system that it needs. In this way, no single process is required to compute

the full global system, saving time and memory. This is the method used by Diffpack.

Grids can be partitioned with respect to grid points, or with respect to elements. In FEM, it is common to partition by elements, since elements are the basis of the grid. See Figure 3 for examples of a structrued and and unstructured grid, both partitioned with respect to elements.

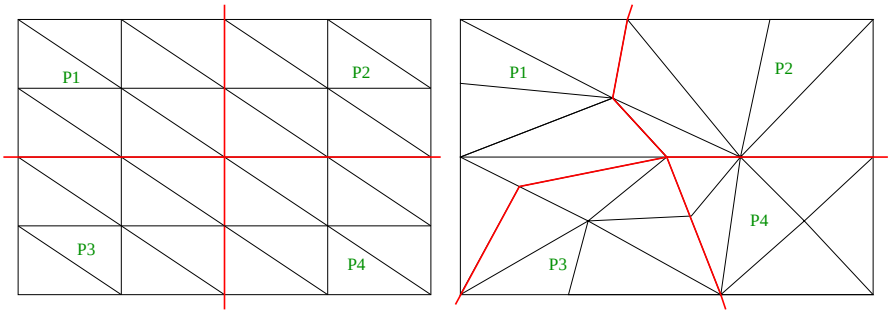

Figure 3: Structured and unstructured grid with four partitions each. Note that the structured grid is partitioned evenly just by dividing it along the axes, whereas a more complicated routine is used to partition the unstructured grid.

1.2.1 Overlap

When the grid is partitioned, the partitions can be extended with a level of overlap. This can be advantageous for parallel preconditioners[5, Ch. 1.6.1]. In Figure 4 (left) a grid is partitioned along the blue line. Both partitions are then extended with one level over overlap, to the red and green lines. Nodes on the boundary of a grid are called boundary nodes. The nodes in a partition that lie on the boundary to another partition are called internal boundary nodes. All other nodes are called interior nodes. The internal boundary nodes in Figure 4 (left) are marked with red and green.

Also, when a grid is partitioned by elements there is bound to be some overlap of nodes, as the partitions are divided along a set of nodes. This set of nodes then belongs to both partitions. See Figure 4 (right) for an example.

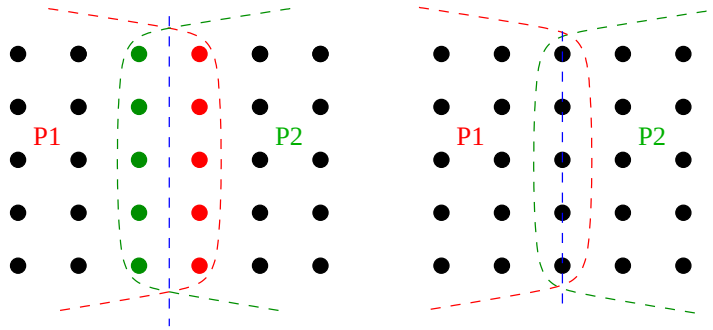

Figure 4: Left: A grid partitioned with respect to grid points (along the blue line), with one level of overlap added (to the red and green lines). The red and green grid points are internal boundary nodes in partitions P1 and P2, respectively.
Right: A grid partitioned with respect to elements (along the blue line). The set of nodes along the blue line now belong to both partitions.

2 Solution of Linear Systems

2.1 Serial Solution of Linear Systems

Most finite element simulations end up as a linear system $Ax = b$ which needs to be solved. The classical way to solve linear systems is by direct solvers, such as Gauss Elimination [10, Appendix C]. However, a large number of iterative solvers have also been invented. These are often superior to standard methods, especially when dealing with sparse matrices, and most linear systems arising from PDEs are sparse [10, Appendix C]. Iterative solvers need only to operate on the non-zeros in the Matrix, and in sparse systems that means a great speedup, in addition to eliminating the problem of fill-in[1].

2.2 Serial Preconditioners

When using iterative solvers, the speed of the solver depends on properties of A. For Krylov subspace methods (often also referred to as "conjugate gradient-like methods"), the rate of convergence usually depend on the spectral properties of A [4, Ch. 2]. Instead of solving $Ax = b$, this system is replaced by $MAx = Mb$, where M is crafted so the spectral properties of MA are better than those of A. If the condition number is reduced, the performance of each iteration improves [3].

2.3 Parallel Solution of Linear Systems

Iterative solvers are well suited for parallel computation. Both the matrix and the vectors can be divided among the processes. This can be done as follows: Assume P processes, and U unknowns and rows in the matrix. Then store U_p vector elements and matrix rows on process p. Now, $U = \sum_p U_p$.

Iterative solvers involve only three linear algebra operations [5, Ch. 1.6.2]:

Vector addition, $\vec{c} = \vec{a} + \vec{b}$. This is a straightforward operation, which involves no communication, since $c[i] = a[i] + b[i]$, and all i-elements are stored in the same process.

Inner product of two vectors $c = \sum_{i=1}^{U} a_i b_i$. This operation can also be completed with very little communication, since the expression can be rewritten as

$$c = \sum_{P} \sum_{j}^{U_p} a_j b_j$$

where j is the local index of the local vector. The inner sum is now a local sum, resulting in a scalar on each process. The only communication needed is in the outer sum, to gather one scalar from each process.

[1] Fill-in means that some entries in the matrix which were originally zero acquire nonzero values during solution of the linear system. This happens e.g. for Gauss Elimination on sparse systems.

Matrix-vector product This operation is the most communication-demanding of the three. Among the local rows, there might be some nonzero-values in columns not corresponding to local points. The corresponding points in the vector will need to be retrieved from their respective processes. Fortunately, in linear systems arising from finite element methods, these points are usually gathered on a small number of processes, due to the way the grid is partitioned. See Section 2.4 and [5] for details.

2.4 Locality of Communication

I will not go into detail about the finite element method here, an introduction can be found in [10]. One detail is crucial to understand the nature of communication in the resulting linear systems, though. All the values in the matrix come from an integrated product of two functions, called a basis function and a weighting function[2]. Both are extremely local, they are zero on all grid points except one. When integrating the product of two such functions, this result will be zero if the two functions do not "belong to" two points very close on the grid. For a one-dimensional example with linear elements, see Figure 5. Again, for more details, please refer to [10].

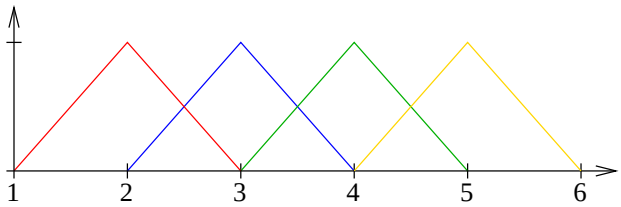

Figure 5: Linear basis functions in a one-dimensional FEM solver. Notice that the product of two functions is zero when they belong to nodes that are not neighbours. For instance, the product of basis function 2 (red), and basis function 3 (blue) is nonzero only in the element between grid point 2 and 3.

This locality of communication proves to be extremely beneficial. Consider a grid point p, residing in partition P. When locality is preserved in the grid partitioning, all grid points that are close to p will belong to the few partitions that are close to P, or P itself. In section 2.3 the distributed matrix-vector was introduced, involving some communication to get the values of vertices laying outside the local partition. Communication is only needed when there are nonzero values in the matrix corresponding to vertices in non-local parts of the vector. And as I have just shown, all these will be found in a small set of non-local partitions. Since all values from one partition can be sent in one message, this means that the number of messages is kept low. This is of high importance, since communication time depends not only on the total size of the data sent, but also on the number of messages[3].

[2] A very brief explanation of how the grid points map to the matrix: No matter the dimension of your grid, number all grid points linearly from 1 to n. The matrix value $A_{i,j}$ is then related to the product of the weighting function i and the basis function j.

[3] Each message introduces some delay due to network latency.

2.5 Parallel Preconditioners

When using parallel solvers, parallel preconditioners are required. Iterative methods for sparse linear systems are easily parallelized (Section 2.3), but preconditioners are not. One method is to run local preconditioners on each process, but this will usually result in a different and possibly weakened effect[5, Ch. 1.6.1]. To compensate for this, a level of overlap (Section 1.2.1) can be added, see [5, Ch. 1.6.1] for details. A completely different approach is to use domain decompositioning methods such as Schwartz methods[5, 2.2], [18, 1.2] and Schur complements[18, 1.2].

3 About Diffpack, Sparskit and pARMS

3.1 About Diffpack

Diffpack is a sophisticated tool for developing numerical software, with main emphasis on numerical solution of PDEs [10]. Diffpack is highly object oriented, and consists of various hierarchies of classes responsible for typical tasks in numerical software. For instance, when programming finite element simulators using Diffpack, the user will write a minimal amount of numerical code, most of the effort goes into combining existing classes, and setting their input parameters. The user is also required to implement small functions that calculate integrals and boundary values. The core of the finite element method, as well as creation, storage and solution of the resulting linear system, is handled by existing Diffpack classes.

3.1.1 Diffpack and Types

Diffpack uses templates, so the classes storing numerical data can be of various types. The two common types are `real` and `Complex`. `real` is not a standard C++ type, but is `#defined` to a standard C++ type. By default it maps to `double`. `Complex` is a Diffpack class handling complex numbers. In this thesis, only real numbers are used, and it is assumed they map to `double`.

3.1.2 Core Classes in Diffpack

Documentation of all Diffpack classes can be found on the world wide web[8], but I will present a few of those most interesting for this thesis here.

Matrix The are a lot of matrix classes in Diffpack, each implementing one storage format. They all inherit from the abstract base class `Matrix`. See Figure 6 for an excerpt of the `Matrix` class hierarchy. In this thesis, `MatSparse`, implementing the Compressed Row Storage[4] (CRS) format, will be used, since matrices arising from FEM simulations are sparse, and CRS is the storage format used internally by Sparskit and for constructing matrices in pARMS.

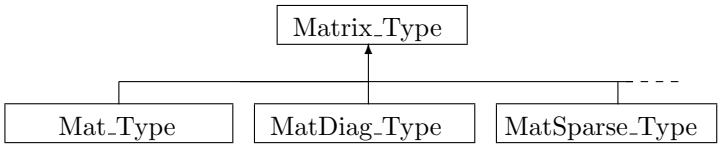

Figure 6: An excerpt of the matrix class hierarchy in Diffpack

Vector The vector hierarchy in Diffpack is quite large, but in this thesis the only class of interest is `Vec`, which implements a one dimensional vector with arithmetic operations.

[4] See Appendix A for more information on CRS.

LinEqSolver Diffpack offers a lot of different linear solvers, both direct and iterative. The iterative solvers are further divided into basic and Krylov subspace solvers. All the solvers have the same interface, inherited from the abstract base class `LinEqSolver`. This allows for runtime selection of the solver class, which can be chosen from a menu. See Figure 7 for an excerpt of the linear solver classes in Diffpack. Having a `Matrix` A, a right hand side `Vector` b, and a `Vector` x, these can be combined in a linear system $Ax = b$, which can be solved by any `LinEqSolver`.

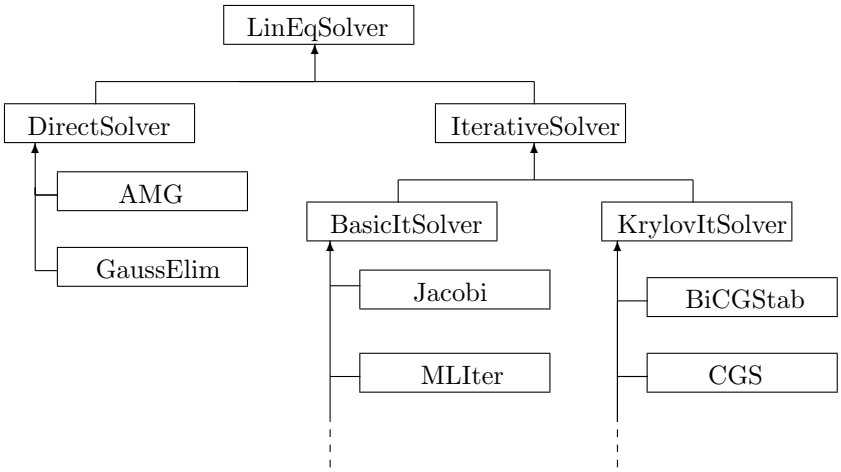

Figure 7: An excerpt of the hierarchy of linear solvers in Diffpack

Precond Diffpack offers a variety of preconditioners, both iterative methods, incomplete factorization, procedural, and multilevel. All preconditioners inherit from the abstract base class `Precond`. See Figure 8 for an excerpt of the `Precond` hierarchy.

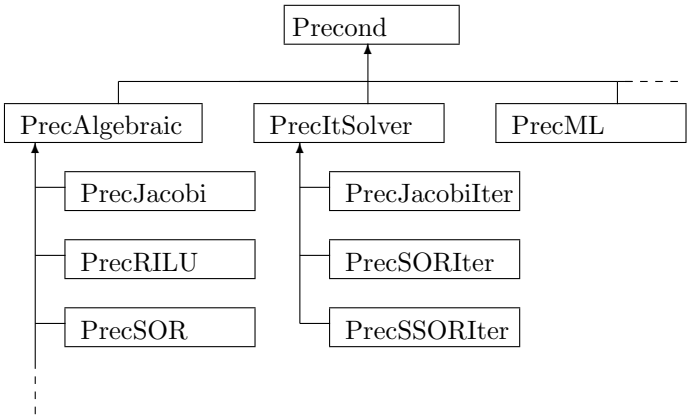

Figure 8: An excerpt of the hierarchy of preconditioners in Diffpack

3.1.3 Parallel Diffpack

A considerable advantage of this object oriented design can be seen when parallelizing code. Diffpack introduces an add-on parallel toolbox [5, Ch. 1.6.2], with minimal impact on existing code. Now the user can reuse a sequential solver that is known to be correct, and parallelize it with only a few extra lines of code. Note for instance that distributed matrices and vectors use the same `Matrix` and `Vector` classes as in serial Diffpack, and that the parallel linear algebra-operations are handled automatically.

3.1.4 Menu System

Diffpack uses a menu system that allows the user to select most parameters at run time. In this way the user can compile the software once, and select options like which solver to use, the grid size, and even the number of dimensions in the problem at runtime. All classes that use information from the menu system have a parameter class responsible for telling the menu system what options are available, and then receiving the users selection. For instance, the class `LinEqSolver` has a parameter class `LinEqSolver_prm` that allows the user to select the solver class at runtime.

3.2 About Sparskit

Sparskit is a basic toolkit for sparse matrix computations[11]. It implements various storage schemes for sparse matrices, and a collection of routines for operations on these. The format used internally by Sparskit is CRS. Examples of routines are storage conversion routines, unary routines like transposition and submatrix extraction, algebraic operations, and (most importantly in this context) linear solvers.

Sparskit is written in Fortran, so integration with C++ based Diffpack is not straightforward. Chapter 4 describes integration of Fortran and C++.

In object oriented programs like Diffpack, the user can create a matrix and two vector objects, and solve this system using a solver object. Sparskit is however not object oriented, and there are a lot more details to take consider. Some of the data structures the user is responsible for storing are the `irow`, `jcol` and `val` arrays of the matrix and work space arrays. Also, the system is not solved by a single call to the `solver()` function. Rather, this function is called once for each iteration, and for each call a vector is returned with a request for the user to do an operation on it. Examples of such operations are multiplication by the coefficient matrix, or testing for convergence.

Sparskit has a number of basic linear solvers, which are listed in Table 3.2. Most of the solvers are also available in Diffpack, but some, like cgnr and dqgmres are not.

Sparskit also offers a range of preconditioners, listed in Table 3.2. These all work with a GMRES solver. A GMRES solver is also available in Diffpack, but the preconditioners are not.

3.3 About pARMS

The parallel Algebraic Recursive Multilevel Solver (pARMS) is a package for solving sparse linear systems on parallel platforms [17]. It has a focus on pre-

Solver	Description
cg	Conjugate Gradient Method
cgnr	Conjugate Gradient Method- for Normal Residual equation
bcg	BiConjugate Gradient Method
bcgstab	BCG stablized
tfqmr	TransposeFree QuasiMinimum Residual method
gmres	Generalized Minimum Residual method
fgmres	Flexible version of Generalized Minimum Residual method
dqgmres	Direct versions of Quasi Generalized Minimum Residual method

Table 1: List of Sparskit basic solvers.

Preconditioner	Description
ilut	A robust preconditioner called ILUT which uses a dual thresholding strategy for dropping elements. Arbitrary accuracy is allowed in ILUT.
ilutp	ILUT with partial pivoting
ilu0	simple ILU(0) preconditioner
milu0	MILU(0) preconditioner

Table 2: List of Sparskit preconditioners for the GMRES solver.

conditioners, and offers variants from both Schwarz procedures and Schur complement techniques.

pARMS has only three linear solvers, shown in table 3.3. Its main focus is however on preconditioners, and a list of available preconditioners is found in table 4. A wide range of the preconditioners do not have equivalent implementations in Diffpack, which is the main motivation for integrating Diffpack and pARMS.

Like Sparskit, pARMS uses the compressed row storage format. It is also not fully object oriented, but it makes heavy use of structs, and makes an effort hiding the details of solving linear systems inside functions. However, it will still need to be wrapped inside new classes to fit into the Diffpack hierarchy.

pARMS itself is written in C. Internally it uses other packages, of which some (such as Sparskit) are written in Fortran. The interface is a pure C library, but Fortran is needed to compile the library. Like Diffpack, pARMS uses MPI [14] for interprocess communication.

Solver	Description
fgmresd	Distributed version of flexible GMRES.
dgmresd	Distributed version of deflated GMRES.
bcgstabd	Distributed version of bi-CG stabilized.

Table 3: List of pARMS linear solvers.

Preconditioner	Description
add_ilu0	Additive Schwarz preconditioner with local ILU0 preconditioner
add_ilut	Additive Schwarz preconditioner with local ILUT preconditioner
add_iluk	Additive Schwarz preconditioner with local ILUK preconditioner
add_arms	Additive Schwarz preconditioner with local ARMS preconditioner
lsch_ilu0	Left Schur complement preconditioner with local ILU0 preconditioner
lsch_ilut	Left Schur complement preconditioner with local ILUT preconditioner
lsch_iluk	Left Schur complement preconditioner with local ILUK preconditioner
lsch_arms	Left Schur complement preconditioner with local ARMS preconditioner
rsch_ilu0	Right Schur complement preconditioner with local ILU0 preconditioner
rsch_ilut	Right Schur complement preconditioner with local ILUT preconditioner
rsch_iluk	Right Schur complement preconditioner with local ILUK preconditioner
rsch_arms	Right Schur complement preconditioner with local ARMS preconditioner
sch_gilu0	Distributed ILU0 preconditioner on interface nodes
sch_sgs	Distributed Gauss-Seidel preconditioner on interface nodes

Table 4: List of pARMS preconditioners.

4 Calling Fortran from C++

Diffpack is written in C++, Sparskit is written in Fortran, and pARMS is written mostly in C, but includes parts written in Fortran. Mixing code written in different languages is not necessarily trivial. As C is a subset of C++, mixing these languages is straightforward. Calling Fortran from C++ is not very hard either, but there are some important issues one needs to be aware of. This chapter discusses some of the most important issues of mixing C++ and Fortran.

4.1 Datatypes

When passing data, one needs to be sure that all systems that handle the data use the same data types. If C++ sends a 64bit floating point number, it is crucial that Fortran is not expecting a 32 bit number etc. In C++, floating point numbers are specified as either single or double precission, `float` and `double` respectively. In Fortran, the corresponding datatypes are `REAL` and `DOUBLE PRECISION`. According to the IEEE Standard for Binary Floating-Point Arithmetic [1], single precision means 32-bit, and double precision means 64-bit. Unfortunately, this is in practice implementation specific, and can not be counted on. The same problem exists with integers. Fortran also provides a way to specify data types with a specific bit-length, using constructs like `REAL*8` which specify an 8 byte long (64 bit) floating point number. There is no way to do this in C++.

C++ also has a lot of advanced data types like structs, enums, classes and function pointers, which are not available in Fortran. Fortran too has some of its own, like common blocks, which are not available in C++.

The only data types that can be safely and portably used are [2]:

C++	Fortran
int	INTEGER
float	REAL
double	DOUBLE PRECISION

In addition to the three basic data types above, it is possible to pass arrays of these types. There are however some critical issues to be aware of.

4.1.1 Indexing

Fortran indexes arrays from 1, whereas C++ and most modern languages index arrays from 0. It is important to be aware of this when programming in either language. When combining code, however, it can often be ignored. If we for instance have a Fortran subroutine that adds two vectors, and C++ code that sets up the vectors and uses the result, C++ will set up the arrays indexed from 0, and pass them to Fortran which treats them as indexed from 1. In both cases, they are indexed from the beginning to the end, so the C++ programmer does not need to be aware of the Fortran way of indexing. If however, the programmer needs to pass specific indexes from C++ to Fortran, these indexes will need to be increased by one.

4.1.2 Storage Order

Another difference concerning arrays is storage order in multidimensional arrays. C++ uses a row-first storage scheme, while Fortran uses a column-first storage scheme. This can prove to be a major headache, as there is no panacea available to solve this problem. To be able to pass a two dimensional array, like for instance a matrix, one has to either transpose the matrix before passing it, or rewrite the code in one of the two languages to use the storage scheme of the other. In this thesis however, I will use compressed row storage (CRS) for sparse matrices, which consists of three one dimensional arrays, and avoid the problem alltogheter.

4.2 Functions and Subroutines

C++ has a single concept of functions. They can take an arbitrary number of arguments, and return zero or one value. Fortran uses two different concepts, FUNCTIONs that return one value, and SUBROUTINEs that do not return any value. Both are usable from C++ like regular C++ functions.

4.2.1 Function Arguments

In C++, function arguments can be passed either as values, pointers or references. In Fortran, all arguments are passed as references. This means that changes to the value of an argument inside the function is always visibile on the outside. This is also the way to return more than one argument from a function, just like in C++. For an example function prototype, see the end of Section 4.3 on page 23.

4.3 Function Prototypes

When compiling code that calls a function, the compiler needs to know that this function exists and what its interface looks like. This is commonly done in C and C++ code, by inserting the function prototype in the source code before the code is called. A function prototype can look like this:

```
int square(int x);
```

Then, even if the compiler has not seen the implementation of the function yet, it knows the function's interface, and that we promise that it is implemented elsewhere [5]. When linking the program, the call to this function will then be linked to the appropriate compiled function.

This works fine when all the code is compiled with the same compiler. Problems start to occur when using different compilers. Each compiler has its own rules for name mangling. These rules dictate what the function's name will be in the object file. A C compiler, not using name mangling, would simply generate the name `square`, whereas a C++ compiler would generate something like `__i__sqr_4i` [9] to allow for function overloading.

Even though Fortran does not support advanced features like templates and function overloading, it, too, needs name mangling because of its case insensitivity. Function and subroutine names must be converted to canonical case.

[5] Either later in the file, or in another file that we will link to.

g77, among others, converts all names to lowercase and appends an underscore. Other compilers convert to uppercase, and some do not append the underscore[16].

So, to be able to call Fortran functions and subroutines from C++, one needs to avoid C++'s name mangling, and in our case make sure the function prototype is written in lowercase, with an underscore at the end. As an example, we want to call the following Fortran function from C++:

```
INTEGER FUNCTION SQUARE(N)
  SQUARE = N * N
  RETURN
END
```

The function prototype in C++ is then:

```
extern "C" int square_(int*);
```

Here, `extern "C"` tells the compiler to switch of name mangling, and the function name is written in lowercase with an underscore appended, to match the object name of the Fortran function.

A similar example, using a SUBROUTINE:

```
SUBROUTINE SWAP(X, Y)
  REAL X
  REAL Y
  REAL Z
  Z = X
  X = Y
  Y = Z
END
```

The function prototype in C++ is then:

```
extern "C" void swap_(float*, float*);
```

Notice that pointers to the numbers are passed, since Fortran has no concept of pass-by-value.

5 Integrating Diffpack with Sparskit and pARMS

5.1 Object Orientation

One of the main purposes of this thesis is to integrate the Sparskit and pARMS linear solvers with Diffpack. Not only should it be possible to use the three programs together, but the details of the Sparskit and pARMS code should be completely hidden from the Diffpack user. This is achieved by creating classes in C++ that wrap around the Sparskit and pARMS code. By doing this, I encapsulate the foreign code, and move from procedural code to a low level of object orientation.

However, object orientation has a lot more to offer. By letting all the solvers inherit from common base classes, say `SparskitSolver` and `pARMSSolver`, code using these solvers can easily change solvers using polymorphism. Diffpack already has an existing hierarchy of solvers utilizing polymorphism, see Figure 7. All Diffpack solvers inherit from `LinEqSolver`. They are then divided into direct and iterative solvers (`DirectSolver` and `IterativeSolver`). The iterative solvers are further divided into basic and Krylov subspace solvers (`BasicItSolver` and `KrylovItSolver`).

To be able to freely interchange Sparskit/pARMS and standard Diffpack solvers, the new solvers need to be fitted into this hierarchy. As a minimum, they should inherit from `LinEqSolver`. However all the new solvers are iterative Krylov subspace solvers, so they could also inherit from either `IterativeSolver` or `KrylovItSolver`. There are now three different places in the hierarchy where the new solvers could fit in. Which one is right?

`IterativeSolver` has some special functions that only apply to iterative solvers, like `performance()` which reports performance-related statistics (convergence statistics etc.). If a user is using these functions in his existing code, he will expect them to still be available when changing to a Sparskit or pARMS solver, since they are also iterative. Hence, they should all inherit from `IterativeSolver`.

`KrylovItSolver` has some functions that are specific to Krylov subspace solvers, mainly related to the residuals (there are no residuals in the basic iterative solvers). These Krylov specific functions should also be available in the Sparskit and pARMS solvers, so they should all inherit from `KrylovItSolver`.

By letting all the new solvers inherit from `KrylovItSolver`, they also inherit from `IterativeSolver` and `LinEqSolver`, and end up inheriting all the wanted properties discussed above. A diagram showing the new classes integrated into the Diffpack hierarchy of solvers is found in Figure 9. Now, pARMS and Sparskit solvers can be used as regular Diffpack solvers.

Another important purpose of this thesis is to integrate pARMS preconditioners in Diffpack. The same issues apply as for the solvers, and more about how this was done can be found in section 5.4.7.

5.2 Getting Raw Data from Linear Systems in Diffpack

Both Sparskit and pARMS will need to get raw data from `Vector` and `Matrix` objects, so I will discuss how to do this here before going into detail about integration with Sparskit and pARMS.

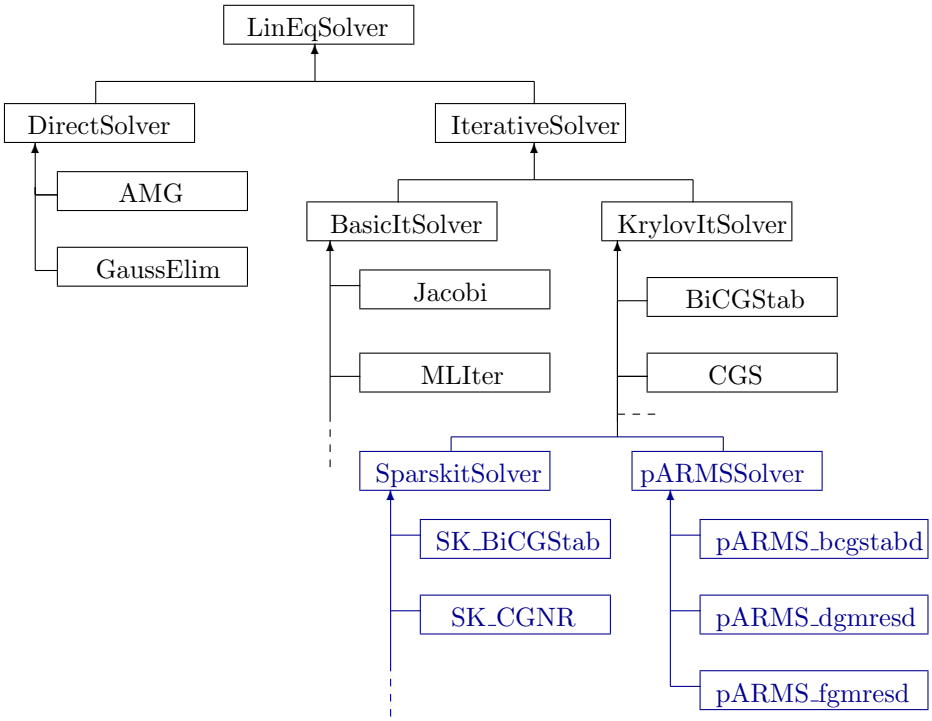

Figure 9: The hierarchy of linear solvers in Diffpack, with the new solvers shown in blue

5.2.1 Getting Raw Data from `Vector` Objects

Getting data out of a `Vector` is as simple as getting a pointer to the `real` array. This is done by calling the `getPtr0()` function.

5.2.2 Getting Raw Data from `Matrix` Objects

Getting the data out of a `Matrix` is harder. First of all, the internal structure of the various subclasses is different. However, Sparskit uses compressed row storage (CRS, see Appendix A) internally, and pARMS matrices can only be constructed using the CRS format, so I can require that this format is used in Diffpack when Sparskit and pARMS integration is used. In Diffpack, the matrix class using CRS is called `MatSparse`.

`MatSparse` has a helper class, `SparseDS` that holds info about the structure of the sparse matrix. An object of this class can be obtained by calling `MatSparse::pattern()`. Pointers to the row and column information can then be obtained from `SparseDS::getIrowPtr0()` and `SparseDS::getJcolPtr0()`. A complete example:

```
1: Handle(Matrix(real)) A;
2: A = new MatSparse(real)(M);
3: MatSparse(real)* SA = (MatSparse(real)*)A.getPtr();
4: SparseDS& s = SA->pattern();
5: int* row = s.getIrowPtr0();
```

```
6: int* col = s.getJcolPtr0();
```

On line 1, a handle to a `Matrix` is declared. Then, on line 2, this handle is set to point to a new `MatSparse` (M is just an object used to initialize the values of A). On line 3, a `MatSparse` pointer to the matrix is obtained, to be able to use functions that are specific to sparse matrices, and not defined on `Matrix`. On line 4, the `SparseDS` helper object is obtained. On lines 5-6, pointers to the row and column information are retrieved.

Now the only remaining issue is how to get a pointer to the `real` array that holds the actual values in the matrix. There is no function available for this. One obvious solution to this problem would be to augment `MatSparse` with a function returning a pointer to this array, say `MatSparse::getDataPtr0()`. According to the requirements for this work, I am not allowed to change existing code in Diffpack, only to write new code, so this approach must be discarded.

When unable to change a class, it is often a good idea to extend it using inheritance. Unfortunately, the data in question is private (not protected), so this approach is unusable.

When augmentation and inheritance is out of the question, another idea is to make a completely new implementation of sparse matrices using CRS. This would involve hundreds, if not thousands, of lines of new code, almost all of it just being duplicated from `MatSparse`. This is clearly not an option.

The method I ended up using is not very pretty, but it beats the other methods by being at all possible, and also small and easy. A pointer to the data array can be obtained using `MatSparse::operator()`. This operator accesses the internal data array in the sparse matrix. Augmenting Example 5.2.2 to retrieve a pointer to the data:

```
6: real* entries =  &SA(1);
```

5.3 Integrating Diffpack and Sparskit

In Section 4, integration of Fortran and C++ was discussed. Here I will discuss the specifics of how to pass data between Diffpack and Sparskit.

5.3.1 Data Types

In Section 4.1, data types in C++ and Fortran were discussed, and a solution was suggested, to stick to corresponding data types like `int`/`INTEGER` and `float`/`REAL`. This is a good idea when starting a project from scratch, but in this case the data types have already been decided on by Sparskit and Diffpack. Sparskit uses `REAL*8`, a specific bit-length data type. Diffpack lets the user choose what data type to use, but the standard is to use `real`(not to be confused with Fortrans `REAL`), a user defined data type which can be specified in a header file. By default it maps to `double`.

Not only can the user choose what type to use with templates, but even if he chooses the standard type, the precision of this type can be changed. In addition to all this, the sizes of C++'s data types are implementation specific. Thus, the user needs to be careful in choosing data types.

5.3.2 What Must be Passed

The goal of a linear solver is to take a matrix and a right hand side, solve the system, and return the solution. But as discussed in Section 3.2, the Sparskit solvers never see the matrix, but rather return a vector and request it to be multiplied with the matrix in question. This enables us to actually do all matrix-vector multiplies in Diffpack, and never let Sparskit see the matrices at all. Sparskit has its own function (`amux`) for matrix-vector multiplies which only handles CRS. This prevents the use of dense matrices in the solvers. Using Diffpack for the multiplies rids us of this limitation. There is one problem with this, though. Sparskit solvers of course do not provide a `Vector` object, but rather a `real` array. This must then be converted to a `Vector` for each matrix-vector multiplication.

Handling dense matrices is outside the scope of this thesis, but an interesting extension would be to convert `real` arrays into `Vector`s in $O(1)$ time, or at least very fast compared to the matrix-vector multiply itself, and see how Diffpack multiplies compares to Sparskit ones in speed and memory usage.

In the rest of the thesis I will let Sparskit handle the multiplies, so I need to get the "raw" data from both `Vector`s and `Matrix`es. See Section 5.2 for more on this matter. Note that when extracting the irow and jcol pointers from the CRS data structure, one would expect them to need to be translated from C++ base 0 to Fortran base 1. But since Diffpack uses base 1 even though it is written in C++, this is not the case, and these arrays can be passed unmodified.

5.3.3 New Classes

The Sparskit solvers are fitted into the Diffpack hierarchy as described in Section 5.1, and depicted in Figure 9.

5.3.4 Symbol Names

When compiling and linking two different programs, chances are that some of the symbol names will collide. This is actually the case with Diffpack and Sparskit, where for instance the symbol name `cg_` collides. There are several ways to solve this.

One way is to patch Sparskit, and change all function names by prepending a prefix. This will change all names from for instance `oldname` to `SK_oldname`. Care must be taken when choosing the prefix to be absolutely sure that no collisions occur. This is a very safe solution, but it requires a lot of work. The patches will take time to write, and they will need to be maintained across versions. Also, this will complicate matters for the users, as they will not only need to install Sparskit, but also patch it before using it.

Another way to fix this would be if the compiler could automatically add the prefix when compiling. Unfortunately, this is not possible with gcc[6].

A compromise is to use the gcc option `-fleading-underscore`, which prepends an underscore to all function names. This will reduce the likelihood of symbol name collisions, but is not as safe as using a longer prefix. This solution is used in the integration of Diffpack and Sparskit.

[6]When nothing else is specified, "gcc" is used to describe the GNU Compiler Collection (including g77 and g++), not the specific GNU C compiler command `gcc`.

5.4 Integrating Diffpack and pARMS

5.4.1 What Must be Passed

Just like with Sparskit, data from the linear system must be extracted. This time however, since the system is distributed, it is also necessary to pass partitioning data. That is, what parts of the linear system belong to what process. The work done in this thesis only support scalar PDEs. I assume that Diffpacks default "special numbering" is used. This means that equation i in the linear system belongs to node i in the grid. When constructing information for pARMS about what equation belongs to what process, information from the grid partitioning can be used.

5.4.2 Overview

Diffpack does not support non-overlapping grids with respect to nodes (See Section 1.2.1), whereas pARMS needs a non-overlapping linear system[7]. Grids with overlap in nodes lead to overlapping linear systems, so the overlap had to be removed. One way to get rid of the overlap would be to rewrite the partitioning system in Diffpack. This would however require significant changes to Diffpack, and the idea is not pursued further. A less invasive procedure is chosen where an existing partitioning scheme is used, and the resulting partition is modified before being passed to pARMS.

The idea is as follows. Partition the grid in the usual way, and use one level of overlap. Then all the internal boundary nodes in one subgrid are interior nodes in some other subgrid. Let each subgrid discard all its internal boundary nodes, and we are left with a grid with no overlap and no orphan nodes. See Figure 10 for a simple example in 2D. This only works under the assumption that there is *exactly* one level of overlap. Diffpack however guarantees that there is *at least* one level of overlap. In [15], Tingstad introduces node based partitioning in Diffpack that fixes this problem and guarantees *exactly* one level of overlap. His software is used for partitioning here.

In the following code, please note that both Diffpack and pARMS index from 1, while C/C++ index from 0. When using regular C/C++ arrays (indexed from 0) I will use the notation `array[i]`, and when using Diffpack vectors (indexed from 1), I will use the notation `vector(i)`.

5.4.3 Discarding internal boundary nodes

To discard internal boundary nodes, the simulator first retrieves an object of the grid partitioner class `GridPart`. This object has two arrays `global_nnrs` and `ib_node_ids`[8]. The former holds the global node numbers of all the nodes for this subgrid. The latter holds the indexes in `global_nnrs` that corresponds to the internal boundary. Now it is just a matter of finding the difference between these two arrays, which is done in Algorithm 1. The indexes of the nodes in `global_nnrs` that are to be kept are stored in `keepTheseNodes`, and the global

[7] When work was started on this thesis, pARMS 3 was used, which does not support overlap. pARMS 3 was revoked during my work, and I had to downgrade to pARMS 2, which actually supports overlap. Read more about this in Section 7.2.

[8] Actually, these are two dimensional arrays, but the interesting information is found in the arrays at `global_nnrs(1)` and `ib_node_ids(1)`.

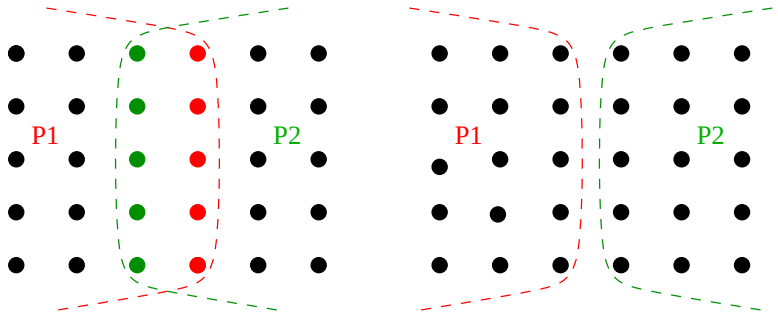

Figure 10: Left, a grid is partitioned with one level of overlap. Note that the green nodes are internal boundary nodes in P2, and interior nodes in P1. Similarly, the red nodes are internal boundary nodes in P1 and interior nodes in P2. Discarding the internal boundary nodes from each subgrid results in the partition to the right, with no overlap.

node numbers for these nodes are stored in `globalIDs`. These two arrays are then used to construct the data structures needed by pARMS. An illustrative example of using Algorithm 1 is shown in Figure 11.

Algorithm 1 Discarding internal boundary nodes to remove overlap

```
for all global_nnrs(1) as i ⇒ nnr do
  if i not in ib_node_ids(1) then
    keepTheseNodes.append(i)
    globalIDs.append(nnr)
  end if
end for
```

5.4.4 Creating mapping information

The distributed vectors and matrices in pARMS need information on mapping from its local nodes to global nodes, and also in which other processes to find non-local nodes. This mapping is represented by two arrays. One array (in my code called `globalMapping`) lists all global node numbers, grouped by, and sorted by, process id. The other array (in my code called `partitions`) contains pointers to where in `globalMapping` each process starts and ends. Process `i` is responsible for global node numbers `globalMapping[partitions[i]]` to `globalMapping[partitions[i+1]]`. This information is already available in `globalIDs` on each process, so constructing these two arrays is simply a matter of gathering them using MPI. See Figure 12 for an example.

5.4.5 Creating the new local matrix

The local matrix is stored using compressed row storage (CRS), see Appendix A. Both `irow`, `jcol` and `val` need to be modified.

`irow` stores the indexes where each row is stored in `jcol` and `val`. This means that the new `irow2` can be built by setting its first entry to 1, and then

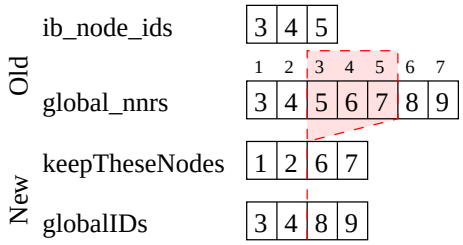

Figure 11: A subgrid has 7 nodes with global node numbers 3-9. Of these, those at index 3-5 in `global_nnrs` (global numbers 5-7) are internal boundary nodes, and discarded.

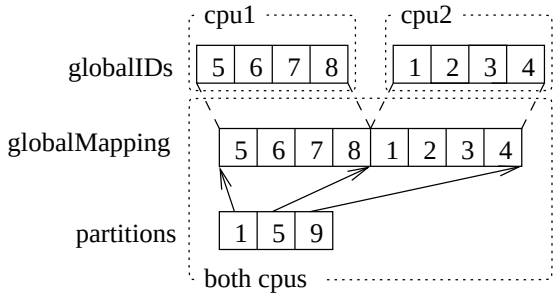

Figure 12: Gathering mapping information. `globalIDs` from each process is concatenated in `globalMapping` on all processes, and `partitions` is created to point to the beginning and end of each process part of this array.

setting each subsequent entry to the previous entry plus the number of values in that row. This is done by Algorithm 2, and illustrated in Figure 13.

`val` stores the value of all the entries in the matrix, and `jcol` stores their corresponding column numbers. `jcol2` and `val2` are constructed by looping over all the rows that are to be kept, and copying the entries in these rows from `jcol` and `val` to `jcol2` and `val2`. This is accomplished by Algorithm 3, and examples are shown in Figure 14.

5.4.6 Creating the new local vector

Creating the new local vector could be as simple as just removing non-local entries in the same way that was done with `jcol` and `val` in the matrix. There is one problem though. Before filling in the entries in the local pARMS vector, the mapping is copied from the local pARMS matrix, so that the vector and the matrix use the same local ordering of variables. Importantly, the order of the local variables used internally in the matrix is not guaranteed to be the same as the one used to create the matrix. Thus the code needs to resort the local vector variables according to the ordering used internally in the matrix.

After having copied the mapping from the matrix to the vector, a property of the vector object, `node[]` is available. Local node `i` now corresponds to global node `node[i]`. There is unfortunately no available mapping from the old

Algorithm 2 Creating the new irow (and count the new number of non-zeroes)

Require: $irow$ = old irow
Require: $noLocalNodes$ = number of local grid points
  $nonZeroes2 \leftarrow 0$
  $irow2[0] \leftarrow 1$
  for $i = 0, i < noLocalNodes$ **do**
    $irow2[i+1] = \backslash$
      $irow2[i] + irow[keepTheseNodes[i]] - irow[keepTheseNodes[i]-1]$
    $nonZeroes2+ = irow[keepTheseNodes[i]] - irow[keepTheseNodes[i]-1]$
  end for

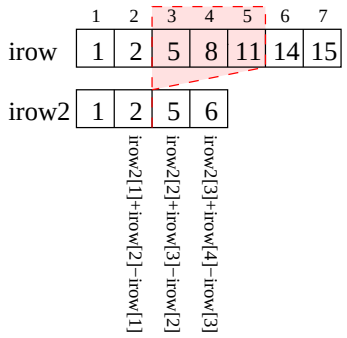

Figure 13: An illustration of Algorithm 2. `irow2` is created based on `irow`, removing rows 3-5.

Algorithm 3 Creating the new jcol and val arrays

Require: $irow$ = old irow
Require: $jcol$ = old jcol
Require: val = old values
Require: $noLocalNodes$ = number of local grid points
  for $i = 0, i < noLocalNodes$ **do**
    rowBase = irow[keepTheseNodes[i]-1]
    for $j = 0, j < irow2[i+1] - irow2[i]$ **do**
      jcol2[irow2[i]+j-1] = global_nnrs(1)[jcol[rowBase+j-1]]
      val2[irow2[i]+j-1] = val[[rowBase+j-1]]
    end for
  end for

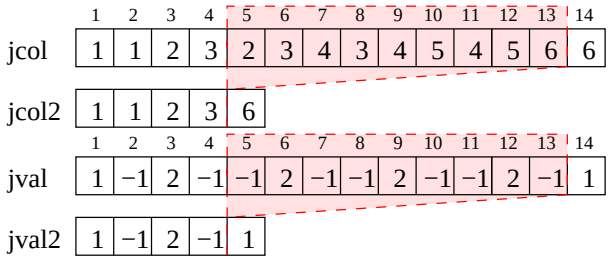

Figure 14: An illustration of Algorithm 3. Just remove all entries belonging to discarded rows.

local index[9] to the new local index[10]. Creating the new `rhs2` from the old `rhs` is solved by Algorithm 4, but the process is a bit complicated, and a textual description of the algorithm follows. See also Figure 15 for an example.

`keepTheseNodes` contains the indexes in `rhs` which are to be kept, sorted by old local index.

`b->node` contains a mapping from new local indexes to global indexes, but is sorted in a different way than `keepTheseNodes`.

`sortedNodemap` is created. This is just `b->node` sorted in the same way as `keepTheseNodes`, so that `sortedNodemap[i]` refers to the same node as `keepTheseNodes[i]`.

`global2local` is created based on `b->node`. This is a mapping from global indexes to new local indexes.

It is now possible to iterate over `rhs[keepTheseNodes]`, and insert the values into `rhs2[global2local[sortedNodemap[i]]]`.

Algorithm 4 Creating the new right hand side vector `rhs2`. See Figure 15 for an example.

Require: rhs = old rhs
Require: $b.node$ = mapping from new local to global index
Require: $keepTheseNodes$ = indexes in `rhs` which are to be kept
  for $i = 0, i < noLocalNodes$ **do**
    sortedNodemap.append(b.node[i])
    global2local[b.node[i]] = i
  end for
  sort(sortedNodemap)
  for $i = 0, i < noLocalNodes$ **do**
    rhs2[global2local[sortedNodemap[i]]] = rhs(keepTheseNodes[i])
  end for

5.4.7 New Classes

Solvers All three pARMS solvers have been integrated into the Diffpack `LinEqSolver` hierarchy, as described in Section 5.1, and depicted in Figure

[9] The order used by Diffpack and when creating the matrix.
[10] The order used internally in the matrix and vector.

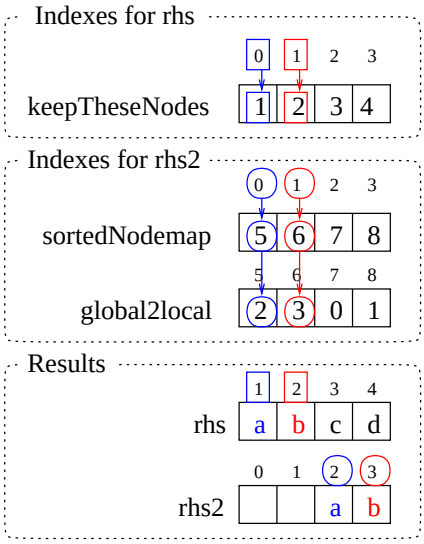

Figure 15: An example of two iterations of the last loop in Algorithm 4. The upper figure shows how the index in `rhs` is found, the middle figure shows how the index in `rhs2` is found, and the result is shown in the lower figure. The first iteration is marked in blue, the second in red. Indexes for `rhs` are marked with a rectangle, indexes for `rhs2` are marked with a circle. a, b, c, d is used to represent data in the right hand side vector. Note that rhs is a Diffpack vector, indexed from 1, whereas the aother arrays are C++ arrays, indexed from 0.

9. See Table 5 for a list of the pARMS solvers, along with a short description.

Class name	Description
pARMSSolver	Parent class for all pARMS Solvers
pARMS_fgmresd	Distributed version of flexible GMRES.
pARMS_dgmresd	Distributed version of deflated GMRES.
pARMS_bcgstabd	Distributed version of bi-CG stabilized.

Table 5: List of pARMS solver classes introduced to Diffpack

Preconditioners All 14 pARMS preconditioners have been integrated into the Diffpack `Precond` hierarchy. See Table 6 for a list of the pARMS preconditioner classes, along with a short description. Figure 16 shows a selection of the new preconditioners, and how they are fitted into the `Precond` hierarchy. The pARMS preconditioners can be used both with pARMS solvers and Diffpack solvers.

Parameter classes To be able to freely interchange Diffpack and pARMS solvers using Diffpacks menu system, as described in Section 3.1.4, parameter classes have been created for both `pARMSSolver` and `PrecPARMS`, `pARMSSolver_-`

Class name	Description
PrecPARMS	Parent class for all pARMS preconditioners
PrecPARMSAddIlu0	Additive Schwarz preconditioner with local ILU0 preconditioner
PrecPARMSAddIlut	Additive Schwarz preconditioner with local ILUT preconditioner
PrecPARMSAddIluk	Additive Schwarz preconditioner with local ILUK preconditioner
PrecPARMSAddArms	Additive Schwarz preconditioner with local ARMS preconditioner
PrecPARMSLschIlu0	Left Schur complement preconditioner with local ILU0 preconditioner
PrecPARMSLschIlut	Left Schur complement preconditioner with local ILUT preconditioner
PrecPARMSLschIluk	Left Schur complement preconditioner with local ILUK preconditioner
PrecPARMSLschArms	Left Schur complement preconditioner with local ARMS preconditioner
PrecPARMSRschIlu0	Right Schur complement preconditioner with local ILU0 preconditioner
PrecPARMSRschIlut	Right Schur complement preconditioner with local ILUT preconditioner
PrecPARMSRschIluk	Right Schur complement preconditioner with local ILUK preconditioner
PrecPARMSRschArms	Right Schur complement preconditioner with local ARMS preconditioner
PrecPARMSSchGilu0	Distributed ILU0 preconditioner on interface nodes
PrecPARMSSchGilu0	Distributed Gauss-Seidel preconditioner on interface nodes

Table 6: List of pARMS preconditioners introduced to Diffpack

`prm` and `PrecondPARMS_prm` respectively. These classes extend `LinEqSolver_prm` and `Precond_prm` in order to enable the new solvers and preconditioners in the menu system.

`pARMSData` This class converts data between Diffpack and pARMS and can hold a matrix converted to pARMS format. This class is used internally by the integrated pARMS solvers and preconditioners, and is never seen by the user.

`pARMS` This class eases integration of pARMS in a Diffpack solver. Including the header file `pARMS.h` makes this class available, and introduces a global object `pARMS parms`, holding global data needed by pARMS. It also introduces a function `void initPARMS()` into the global scope. This function makes pARMS solvers and preconditioners available in the menu system. The `pARMS` class functions are:

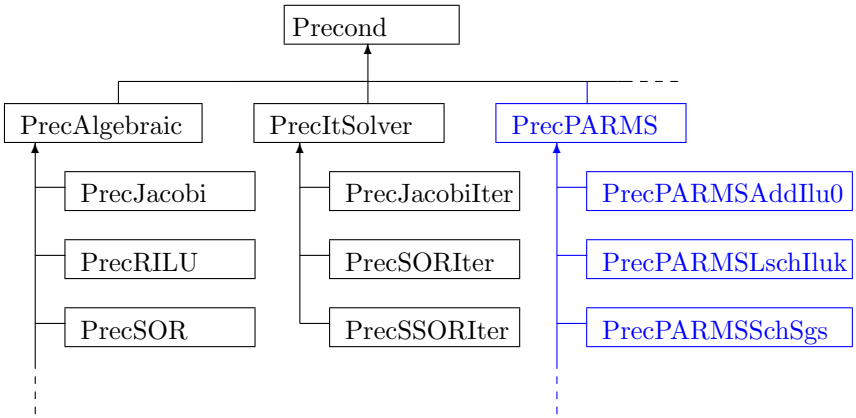

Figure 16: The extended hierarchy of preconditioners in Diffpack, with the new ones shown in blue

`void attachCommAdm( const SubdCommAdm&)`	Attach a SubdCommAdm object to the pARMS object, needed by the pARMS solvers and preconditioners.
`SubdCommAdm& getSubdCommAdm()`	Get the SubdCommAdm object attached in the above function.

For a practical guide on how to use pARMS in a parallel Diffpack solver, see Appendix D.

6 Application Examples with pARMS

There are two goals in this section. A range of new solvers and preconditioners have been introduced to Diffpack, which can be used to solve an arbitrary scalar PDE. The first goal of this section is to confirm that these new solvers and preconditioners can be used as described in Section 5.1, and that they indeed converge.

Second, a comparison of Diffpack and pARMS solvers and preconditioners will be done. The convergence performance of any linear solver and preconditioner is highly problem dependent, so deciding which is best[11] is a huge task. This is outside the scope of this thesis, but a comparison will be made using two model problems. The first is a time-independent Poisson equation, the second is a time dependent shallow water wave simulator using the Boussinesq approximation. Both problems are solved with three combinations of solvers and preconditioners:

1. Diffpack solver with Diffpack preconditioner
2. Diffpack solver with pARMS preconditioner
3. pARMS solver with pARMS preconditioner.

These experiments will not be sufficient for a general rating of the solvers and preconditioners, but will provide some useful insights into the computational performance of the Diffpack/pARMS integration. More specifically, the relative time spent on storage conversion, and the relative time spent on linear solvers in the simulators will be studied.

6.1 Hardware and compilation

All benchmarking is done on the cluster `chilopodus.simula.no`. It consists of 24 nodes, each with the specifications listed in Table 6.1.

CPUs	2x Itanium2 1300 Mhz
Cache	16 KB L1, 256 KB L2, 3 MB L3
RAM	4 GB shared
Operating System	Debian GNU / Linux 3.1 “Sarge”
Kernel	Linux ia64 SMP 2.6.8
Network Interface	Gigabit Ethernet
MPI Implementation	MPICH 1.2.6
Queue System	Torque 1.2.0 (p0)

Table 7: Specifications for each node on the cluster `chilopodus.simula.no`.

pARMS was compiled using the makefile in Appendix C, using gcc[12] and ifort, both with the optimization option `-O3`. Diffpack was compiled using the makefile `.cmake2` in Appendix C, using the optimization option `Make mode=OPT` (which includes setting `-O3`).

[11] Whatever “best” might mean. Low computational time, low memory usage and good scalability are common indicators.

[12] The makefile says mpicc, which is a symlink to cc, which is a symlink to gcc

6.2 Measuring time

To measure the performance of the solvers and preconditioners I introduce a class `SolverTimer`. This class is a state machine, with the states listed in Table 8.

State	Description
ST_SETUP	Time spent setting up the system, everything that happens within `solveProblem()`, but outside `lineq.solve()`. Examples are assembling the matrix and inserting boundary values.
ST_MATRIX	Time spent converting the coefficient matrix from Diffpack to pARMS, as described in 5.4.5.
ST_VECTOR	Time spent converting the right hand side vector from Diffpack to pARMS, and the solution back again. As described in 5.4.6.
ST_MAPPING	Time spent converting the mapping from Diffpack to pARMS, as described in 5.4.4
ST_PRECON_INIT	Time spent setting up pARMS Preconditioner.
ST_PRECON_CONV	Time spent converting vectors from Diffpack solvers for application of pARMS preconditioners.
ST_SOLVE	Time spent in the solver
ST_PRECON_APPLY	Time spent applying pARMS preconditioner to Diffpack solver.
ST_OTHER	Time spent on other integration issues.

Table 8: States supported by `SolverTimer`

A global object `solverTimer` is initialized, and is then used throughout the program by calling `solverTimer.setState(ST_STATE)` each time the program enters a new state. At the end of the program, the time spent in each state is dumped to file. This provides a detailed view into how time is spent during the computations, but for the bigger picture, some of the states are combined as in Table 9.

6.2.1 Defining and Grouping States

Some comments about how the states were chosen and grouped: First, the time spent creating and scanning the menu, generating reports etc., are not included. The interesting part is the time spent actually solving the problem. Measuring starts when `solveProblem()` is called. Inside this function, the linear system is set up, essential boundary conditions are filled in, and the resulting system is solved. For a more advanced simulator, extra calculations can also be done here. Examples are preparing for the next time step, calculating derived fields etc. Except for the actual solution of the linear system, everything in this function is reported as `ST_SETUP`, which is also the sole component of the group "Setup".

The next group, "Conversion" encompasses all conversion that is done once for each solve. This means converting the coefficient matrix, the mapping, the right hand side and solution vector, setting up the preconditioner, and some other minor calculations.

State	Composed of	Description
Setup	ST_SETUP	Time spent setting up the system.
Conversion	ST_PRECON_INIT ST_MATRIX ST_VECTOR ST_OTHER ST_MAPPING	Conversion of data structures in the linear system, and setup of the preconditioner. Done once for each `solve()`
Preconditioner Conversion	ST_PRECON_CONV	Conversion of vectors for application of preconditioner. Done each time the preconditioner is applied.
Solve	ST_SOLVE ST_PRECON_APPLY	Time spent in the solver and preconditioner.

Table 9: Combined states from `SolverTimer`

One conversion issue is left out of the "Conversion" group, namely that of converting vectors for application of the preconditioner. All conversion mentioned above is done once for each solve, but this is done each time the preconditioner is applied, which might pose a bigger problem. The state `ST_PRECON_CONV` is separated in its own group, "Preconditioner Conversion".

The final group is "Solver". This group encompasses both the time in the solver, and in application of the preconditioner. These are actually two states, but they are only possible to distinguish in the combined Diffpack solver / pARMS preconditioner. To ease comparison of the pure and combined solvers, preconditioner application and solver time is summed up for the combined solver too. Also, the time spent in the solver is highly dependent on the efficiency of the preconditioner (and vice versa), so this distinction is not very meaningful in the big picture.

6.2.2 Reporting Time

The problems will be solved on 2, 4, 8 and 16 CPUs. All CPUs are equal, and have access to identical resources, see Section 6.1. Thus, for perfect speedup, one would expect a doubling of CPUs to reduce computational time by 50%. In the following, I will report the sum of time spent on all CPUs. Given perfect speedup, reported time is then constant, regardless of the number of CPUs. Now, scalability will be apparent in the graphs. This also has the added benefit that the graphs will be of approximately the same height, and avoiding the problem of details being obscured by huge differences in graph heights.

6.3 Selection of Solvers and Preconditioners

A multitude of solvers and preconditioners are now available in both Diffpack and pARMS. As solvers, Diffpacks `BiCGStab` and pARMSs `bcgstabd` were chosen, since they are both implementations of bi-conjugate gradient stabilized method, and should be expected to perform quite similarly. The only other method implemented by both packages is the generalized minimal residual method, which is reported to be "not optimal" in its current implementation

in Diffpack[7].

All applicable preconditioners were tried for each combination of solver and problem, and were selected for use in the final experiments based on good performance. Again, the goal is not to benchmark the preconditioners, but to gain insights into how time is spent in the program. The preconditioners selected are presented in Table 10.

6.4 Convergence

Diffpack supports several convergence monitors, while pARMS has only one. The pARMS stopping criterion is $\|current\ residual\|/\|initial\ residual\| < tolerance$. This is the same criterion as in Diffpacks `CMRelResidual`, so this convergence monitor is used in all experiments. The tolerance in both monitors is set to $1.0e-4$, the Diffpack default.

	Poisson1	Boussinesq
Diffpack	RILU	SOR
Combined	LschIluk	RschIluk
pARMS	LschIluk	RschIluk

Table 10: Preconditioners used in the benchmarks

6.5 Poisson Equation

The first problem studied is a two dimensional time-independent Poisson equation. This equation is solved using 2D linear triangle finite elements on a 1000 x 1000 grid. The simulation is run on 2, 4, 8 and 16 CPUs, and time is reported as described in Section 6.2.2. See Figure 17 and Table 11 for the results from this simulation.

#	Method	Setup	Conv	P. conv.	Solver	Total	Iters	Sec/I
2	Diffpack	10.9	0.0	0.0	102.2	113.1	81	1.3
	Combined	10.8	9.6	9.6	66.0	96.0	11	6.0
	pARMS	10.9	12.5	0.0	70.0	93.5	11	6.4
4	Diffpack	10.9	0.0	0.0	118.9	129.8	94	1.3
	Combined	11.0	12.1	11.5	89.4	124.0	14	6.4
	pARMS	10.8	15.0	0.0	93.7	119.4	14	6.7
8	Diffpack	11.0	0.0	0.0	126.9	137.9	101	1.3
	Combined	11.1	17.0	11.5	106.9	146.5	14	7.6
	pARMS	11.1	19.6	0.0	110.4	141.1	14	7.9
16	Diffpack	11.3	0.0	0.0	120.2	131.5	92	1.3
	Combined	11.0	21.6	12.1	123.1	167.7	15	8.2
	pARMS	10.9	24.8	0.0	126.9	162.6	15	8.5

Table 11: Statistics from the Poisson simulator on 2, 4, 8 and 16 CPUs. See Table 9 for a description of the states "Setup", "Conversion", "Preconditioner Conversion", "Solver" and "Total". "Iters" is the number of iterations used, "Sec/I" is the number of seconds used (in the state "Solver") per iteration.

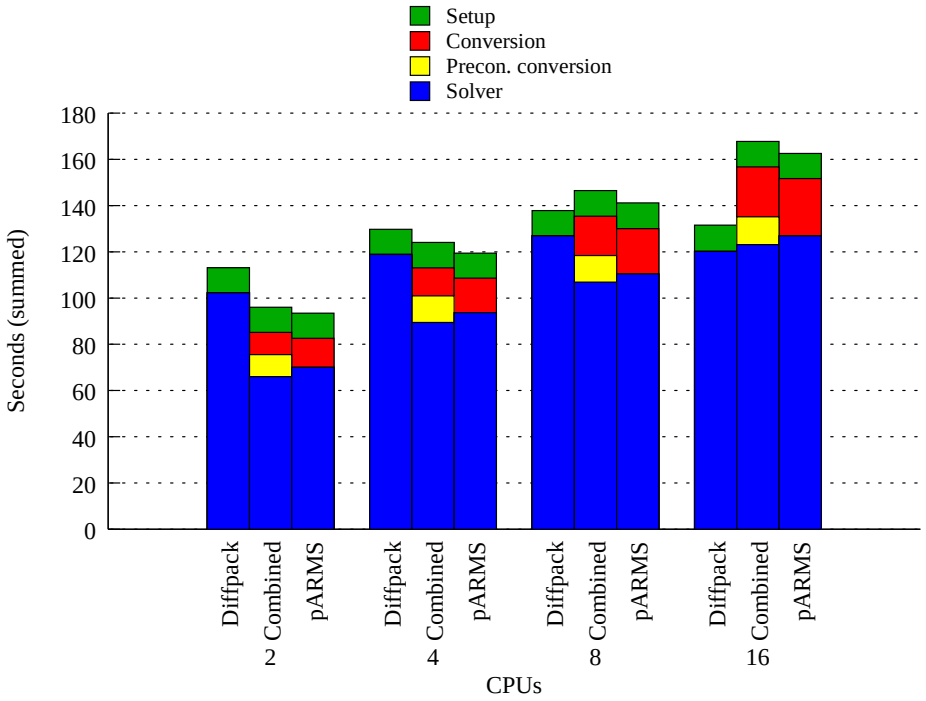

Figure 17: Summed runtime for the Poisson simulator on 2, 4, 8 and 16 CPUs. The pARMS Schur preconditioner is superior when the relative amount of internal boundary nodes is low.

In the case of 2 and 4 CPUs, the Diffpack solver is notably more effective when using the pARMS LschIluk preconditioner than when using the Diffpack RILU preconditioner. For 2 CPUs, time in the solver is reduced by as much as 35%, and total time is reduced by 17%. However, using 8 CPUs, the advantage is overshadowed by conversion time, and for 16 CPUs, the RILU preconditioner is faster even disregarding conversion time.

In the last column in Table 11, we see that LschIluk spends far more time per iteration than RILU. When calculating this fraction, only time spent in the solver and preconditioner was counted in. All converting time, including preconditioner converting, was kept out. This difference in time per iteration shows that the LschIluk preconditioner is a more demanding procedure than RILU. The number of iterations is also far lower, showing that the LschIluk is a lot more effective per iteration. These results indicate that LschIluk is a more advanced routine than RILU, demanding more time, but at the same time reducing the residual more per iteration.

Both LschIluk and RILU are variations of ILU, but RILU uses it as part of an additive Schwarz procedure, while LschIluk uses it as part of a Schur complement technique. The time per iteration increases with the number of CPUs for LschIluk, but not for RILU. The Schur Complement technique focuses on the interface problem, which size is dependent on the number of internal boundary nodes. When the number of grid points is kept constant, but the

number of partitions is increased, the relative amount of internal boundary nodes is increased. This results in more work for LschIluk, and it does not scale as well as RILU.

6.5.1 Fixed Iterations and Error Comparison

Measuring time and iterations means measuring how long it takes for the solver to be satisfied with the solution, and when the solver is satisfied depends entirely on the convergence monitor. Comparing the pure Diffpack solver and the combined solver is possible, as they both use the Diffpack convergence monitor. Comparisons with the pure pARMS solver is however not as straightforward. It uses a conceptually similar convergence monitor, but the implementation details will always differ.

To be able to compare the Diffpack solver and the pARMS solver, I instead set the convergence tolerance to a very strict value ($1.0e-10$), and limit both solvers to 20 iterations, ensuring that they are both stopped by the maximum number of iterations, not the convergence monitor. The Poisson simulator includes the analytical solution, and computes the L2 norm of the error at the end of the simulation. Table 12 lists both time spent and the error in the solution. The time spent is also shown in Figure 18 (top), together with the error (bottom). All experiments used 20 iterations.

#	Method	Setup	Conv	P.conv.	Solver	Total	Sec/I	Error
2	Diffpack	10.8	0.0	0.0	27.2	38.0	1.4	1.52817
	Combined	10.8	9.6	17.0	119.2	156.6	6.0	0.16526
	pARMS	10.9	12.6	0.0	122.6	146.1	6.1	0.16526
4	Diffpack	10.9	0.0	0.0	27.4	38.3	1.4	1.07303
	Combined	11.0	12.1	16.2	128.4	167.7	6.4	0.24903
	pARMS	10.9	15.0	0.0	132.1	158.0	6.6	0.24907
8	Diffpack	11.2	0.0	0.0	27.4	38.5	1.4	0.74679
	Combined	11.1	16.9	15.6	147.5	191.1	7.4	0.11406
	pARMS	11.1	19.6	0.0	150.4	181.1	7.5	0.11412
16	Diffpack	10.9	0.0	0.0	27.1	38.0	1.4	0.51203
	Combined	10.9	21.2	15.5	159.8	207.4	8.0	0.06278
	pARMS	10.9	24.2	0.0	161.2	196.4	8.1	0.06277

Table 12: Statistics from the Poisson simulator on 2, 4, 8 and 16 CPUs, limited to 20 iterations. See Table 9 for a description of the states "Setup", "Conversion", "Preconditioner Conversion", "Solver" and "Total". "Sec/I" is the number of seconds used (in the state "Solver") per iteration. "Error" is the average L2 norm of the solution on each CPU.

The time per iteration is approximately the same as in the previous experiment, and since the LschIluk preconditioner uses a lot more time per iteration, the combined and pure pARMS solvers use a lot more time than the pure Diffpack solver. This is as expected. What is interesting in this experiment is the error. Using LschIluk leads to a much smaller error than using RILU. This supports the finding in the previous experiment, confirming that LschIluk spends

more time, and reduces the residual more, per iteration. Also, the poorer scaling reappears.

One new interesting point in this experiment is that Diffpack/pARMS and pure pARMS result in almost exactly the same error. They are using different implementations of the bi-conjugate gradient stabilized method, but the same LschIluk preconditioner. Also, the difference in solver time (excluding all conversion) between the two never exceeds 3%. This indicates that the two solvers are equally well implemented. When looking at the total solver time however, including all conversion, pure pARMS performs better. This is as expected, as it does not need to do vector conversion each time the preconditioner is applied.

6.6 Boussinesq Simulator

In the following experiment, the Boussinesq shallow water simulator from [10, Section 6.2.5] is studied. The simulator is run on 2, 4, 8 and 16 CPUs, with a 200×200 grid. The simulator is time dependent, and solves two linear systems at each time level. Time runs in the interval $[0, 3]$, with time steps of 0.25. For all time steps, the number of iterations and time spent in the two solves are added.

In this simulator, the analytical solution is not available, so it is not possible to analyse the real speed of convergence, as was done in Section 6.5.1.

The results from the simulator are available in Table 13 and Figure 19. Again, the two solvers using the pARMS preconditioner converge in about the same number of iterations, substantially fewer than the pure Diffpack solver, and each iteration uses a lot more time.Also, both solver and total time is shorter when using the pARMS preconditioner, especially for few CPUs. For 2 CPUs, the combined solver is 35% faster than the pure Diffpack solver.

Note that in this problem, with a more complicated `solveSystem()` function, the time in the setup sate is responsible for a bigger part of the total solver time. This means that the significance of a speedup in the solver will be less significant in practice.

#	Method	Setup	Conv	P. conv.	Solver	Total	Iters	Sec/I
2	Diffpack	33.8	0.0	0.0	144.3	178.1	2158	0.07
	Combined	34.1	8.5	13.6	48.0	104.2	32	1.50
	pARMS	33.5	10.0	0.0	72.5	116.0	30	2.42
4	Diffpack	38.4	0.0	0.0	154.2	192.6	2178	0.07
	Combined	37.6	10.9	3.8	69.3	121.7	46	1.51
	pARMS	37.7	11.5	0.0	97.4	146.6	46	2.12
8	Diffpack	48.1	0.0	0.0	134.1	182.3	2190	0.06
	Combined	47.7	13.5	1.8	82.2	145.1	46	1.79
	pARMS	45.1	14.2	0.0	110.6	169.8	46	2.40
16	Diffpack	72.7	0.0	0.0	174.1	246.8	2194	0.08
	Combined	70.9	18.3	2.4	101.7	193.2	49	2.08
	pARMS	71.1	20.7	0.0	134.3	226.1	51	2.63

Table 13: Statistics from the Boussinesq simulator described in 6.6. See Table 9 for a description of the states "Setup", "Conversion", "Preconditioner Conversion", "Solver" and "Total". "Iters" is the total number of iterations used for the two solves in all the time steps, "Sec/I" is the number of seconds used (in the state "Solver") per iteration.

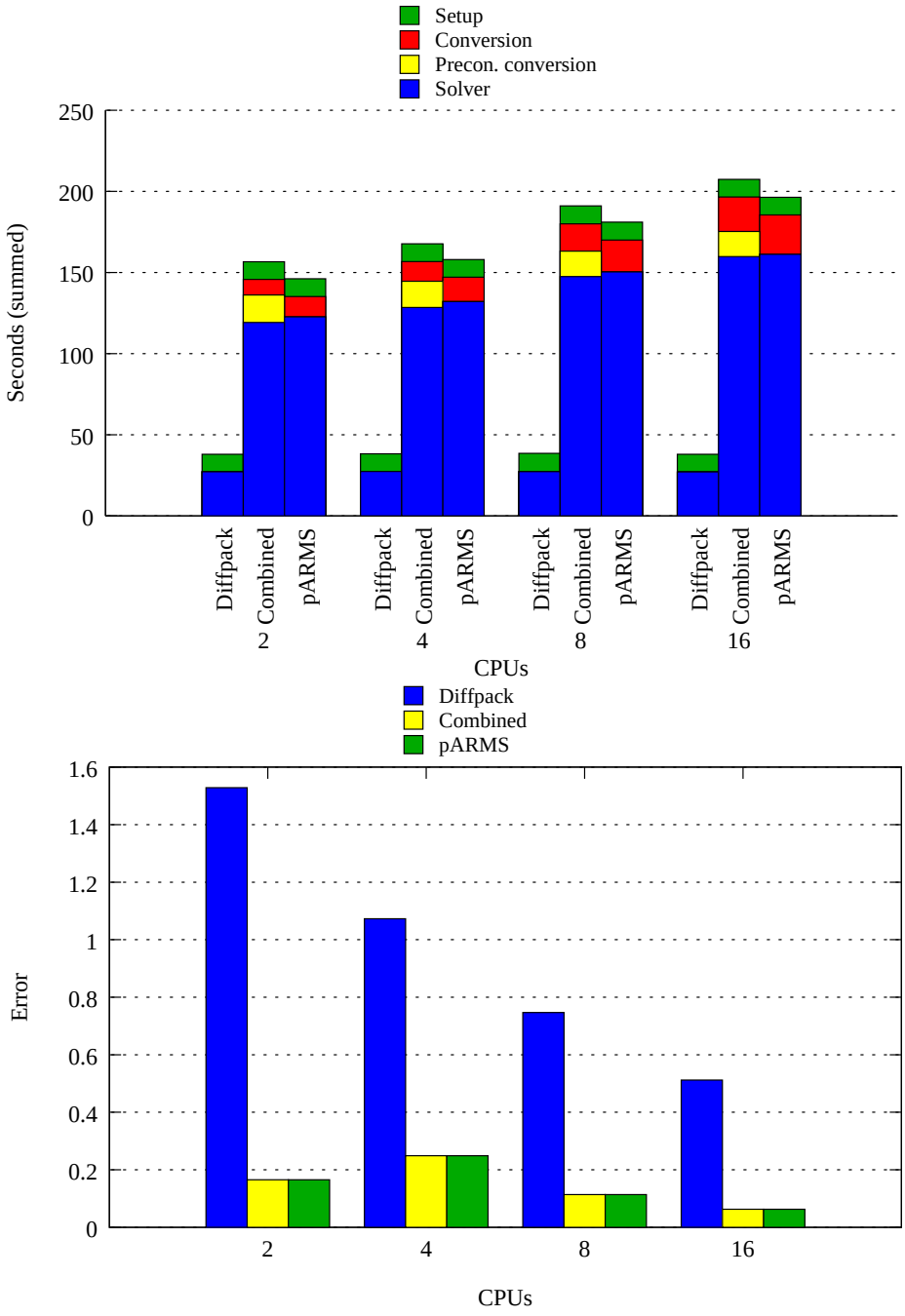

Figure 18: Results from the Poisson simulator on 2, 4, 8 and 16 CPUs, limited to 20 iterations.
Top: Summed runtime for each method.
Bottom: Average L2 norm of the solution on each CPU.
The methods "Combined" and "pARMS" both use a pARMS Schur preconditioner, which uses more time per iteration, while also reducing the residual more. This is discussed further in Section 6.5.1.

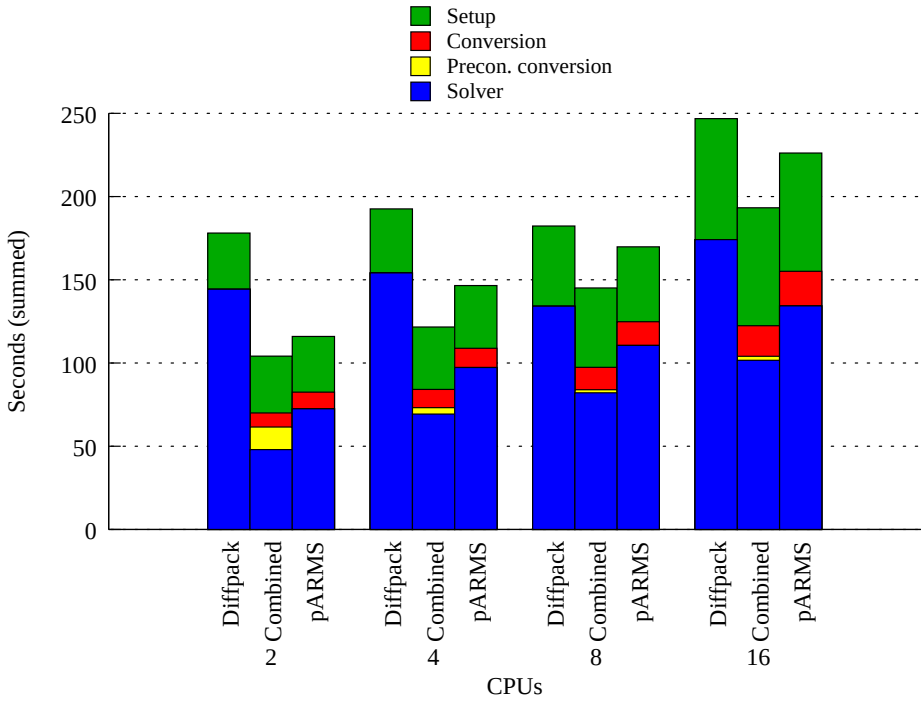

Figure 19: Results from the Boussinesq simulator described in 6.6. The methods "Combined" and "pARMS" both use a pARMS Schur preconditioner, which results in faster convergence. Note also how the time in the setup state (green) is much more significant than for the Poisson simulator in Figure 17.

7 Conclusion and Future Work

7.1 Conclusion

The two main questions raised as goals for this thesis are:

1. Diffpack already has a toolbox for solving linear systems in parallel. Is it possible to extend this toolbox in a flexible way, allowing the Diffpack user to use techniques from pARMS as though they were just another part of Diffpack?

2. What can be gained from this extension? Can the introduction of pARMS make Diffpack simulators more efficient?

The answer to the first question is *yes*. I have shown how to integrate pARMS with Diffpack allowing the user to select pARMS solvers and preconditioners from the menu system. No original Diffpack library files have been changed in the process, and only two additional lines of code are required to extend a Diffpack simulator with pARMS capabilities.

The answer to the second question is *it depends*. pARMS offer a set of powerful preconditioners not found in Diffpack, and for some problems, they can provide a significant speedup. In the Poisson solver in Section 6.5, speedups of up to 17% were found, and in the Boussinesq solver in Section 6.6, speedups as high as 35% were found. However, the speedup is dependent on the problem, the number of CPUs and the grid size. For some cases, using pARMS actually slows down the simulator.

> The introduction of pARMS can make Diffpack simulators notably more efficient for some problems. Thanks to the flexible integration, it is easy for a Diffpack user to test if this is the case for his particular problem.

7.2 Future Work

The pARMS extension only supports scalar PDEs, not vector PDEs. This limitation should be overcome by replacing the mapping routine for the distributed linear system with a more advanced one.

In a time dependent problem, the coefficient matrix is sometimes the same for each time step. By making the pARMS-datastructure a class member of `pARMSSolver` instead of a local variable in the `solve()` routine, it will not have to be calculated for each time step. Furthermore, in some simulators, the coefficient matrix does change, but its structure remains the same. In this case, some of the mapping structures might be preserved between time steps. Both these features must be optional, since some solvers call `solve()` repeatedly, but with different coefficient matrices.

Convergence monitors should be better integrated, i.e. the pARMS convergence monitor should be written as a class in Diffpacks `ConvMonitor` hierarchy, with its own `ConvMonitor_prm` class for parameters.

When work was started on this thesis, pARMS 3 was used, which does not support overlapping rows in the distributed matrix. pARMS 3 was revoked during my work, and I had to downgrade to pARMS 2, which actually supports overlap. At that time, it was too late to change the conversion code. It is

not known if the notion of overlap in Diffpack is compatible with the notion of overlap in pARMS 2. If so, a lot of time converting formats could probably be saved.

A Compressed Row Storage (CRS)

(Please note that some sources refer to Compressed Row Storage (CRS) as Compressed Sparse Row (CSR). The terms are equivalent.)

As with all storage formats for sparse matrices, the goal of CRS is to save storage space and CPU cycles. If most of the $A_{i,j}$ are zeroes, there is no need to store or operate on them. CRS introduces three arrays to store one sparse matrix:

> A real array *val*, with length equal to the number of nonzero entries in the matrix A. Alle the nonzero entries are stored here.
>
> An integer array *jcol*, with the same length as *val*. This stores the column indices of the values in *val*.
>
> An integer array *irow*, with length equal to the number of rows in A, plus one. The values in *irow* stores pointers to the beginning of each row in *val* and *jcol*. The last entry stores where the $n+1$th row would have started. This last entry makes it easier to iterate over the matrix.

Consider the following example:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & & \\ 3 & 4 & 5 & \\ & 6 & 7 & 8 \\ & & 9 & 10 \end{pmatrix}$$

This is a standard banded matrix, which serves as a good example of a sparse matrix. When storing this matrix in CRS format, the following arrays are created:

All the entries in A are stored in *val*, row by row. The column-index of each of these entries are stored in the corresponding cells in *jcol*:

val:	1	2	3	4	5	6	7	8	9	10
jcol:	1	2	1	2	3	2	3	4	3	4

According to these arrays, "1" goes in column 1. "2" in column 2, "3" in column 1, etc. It remains to store where the rows start. This is done in *irow*:

Array index:	1	2	3	4	5
irow:	1	3	6	9	11

By looking at *irow*[1] and *irow*[2], it is clear that the first row in A can be found in *val*[1] to *val*[2]. Row 2 is found by looking at *irow*[2] to *irow*[3], which tells us that it is to be found in *val*[3] to *val*[5]. Generally, row i is found in $val[irow[i]]$ to $val[irow[i+1]-1]$. (The *jcol* indexes are always the same as the *val* indexes.)

The last value of *irow* is special. It tells where the fifth row would have started, if there was one. In this example, the first entry in the fifth row would have started in *val*(11), so *irow*(5) is set to 11. This makes it easy to iterate over the entire matrix, as one can iterate to $irow[i+1]$ without overrunning the array.

B Installing Sparskit

Sparskit can be dowloaded from [13]. It is free software, released under the GNU Lesser General Public License version 2.1 [6].

To install Sparskit, just unpack the tarball and type `make all`.

The following `.cmake2` was used to compile Diffpack with Sparskit:

```
-|.cmake2|-----------------------------------------------------------
APPL := app
LIBS += ../SPARSKIT2/ITSOL/iters.o ../SPARSKIT2/ITSOL/itaux.o\
 ../SPARSKIT2/BLASSM/matvec.o ../SPARSKIT2/UNSUPP/BLAS1/blas1.o\
 ../SPARSKIT2/ITSOL/ilut.o ../SPARSKIT2/FORMATS/formats.o\
 ../SPARSKIT2/FORMATS/unary.o ../SPARSKIT2/BLASSM/blassm.o
-------------------------------------------------------------------
```

C Installing pARMS

pARMS can be dowloaded from [12]. It is free software, released under the GNU Lesser General Public License version 2.1 [6].

Installing pARMS is different for each platform, and makefiles are included for a number of architectures. To get it to compile on chilopodus (Linux ia64 SMP) I used the provided `makefile.inLINUXg77`, but modified it to use Intel ifort 9.0 instead. The following modified makefile was used:

```
-|makefile.in|-------------------------------------------------------
# path for this directory
PARMS_ROOT  = /home/anderkn/master/scratch/PARMS-2.2

# name used for architecture
ARCH    =   LINUX
DARCH   =   -D$(ARCH)

# variable to declare optimization level
# use '-g' to create libparms for debugging purposes
DBG     =      -O3

# archive command
AR  =   ar
ARFLAGS =   cr
#=====================================================================

#=====================================================================
# Options for a generic LINUX configuration
#################################
 CC =   mpicc
 CFLAGS =   $(DBG) $(DARCH)
 INCLUDE_METIS =     -I$(HOME)/metis-4.0/Lib
 METIS_HOME  =       -L$(HOME)/metis-4.0

# fortran compiler / linker
 FC = /simula/software/intel/fc/9.0/bin/ifort
 FFLAGS = $(DBG)  $(DARCH) -I/usr/include -I/usr/lib/mpich/include\
  -assume 2underscores
#
# the directory of MPI library. for example -L/usr/local/mpich/lib
 LFLAGS_MPI     =
# the mpi library
 LIBS_MPI   = -lmpi
# the directory of BLAS
 LFLAGS_BLAS    =
# the BLAS library
 LIBS_BLAS  =   -lblas
#LINKER
# LINKER    =   $(FC)
 LINKER = $(CC)
#LINK OPTION
 LINK_OPT   = -L/opt/intel/fc/9.0/lib -lifcore -limf -lunwind

---------------------------------------------------------------------
```

The following `.cmake2` is used to compile Diffpack with pARMS:

```
-|.cmake2|-----------------------------------------------------------
LDPATH += -L/home/anderkn/master/scratch/PARMS-2.2/LIB\
 -L/opt/intel/fc/9.0/lib
LIBS += -lparms-O3 -lifcore  -limf -lunwind -lblas
INCLUDEDIRS += -I/home/anderkn/master/scratch/PARMS-2.2/INCLUDE\
 -DSOLVERTIMER
-------------------------------------------------------------------
```

See also Appendix D for a guide on how to extend a parallel Diffpack solver with pARMS capabilities.

D How to use pARMS with Diffpack

See Appendix C for instructions on how to download and install pARMS, and how to link Diffpack with pARMS. For a description of the class `pARMS` used below, see the end of Section 5.4.7.

To use pARMS with Diffpack, some modifications need to be made in the `main()` routine, as well as in the simulator class. The starting point is assumed to be a parallel Diffpack simulator. If you are starting with a serial Diffpack simulator, first follow the instructions in [5, Chapter1.6.4].

main In the main routine, after you have called

```
initDiffpack()
```

call

```
initPARMS()
```

which is defined in `pARMS.h`

Simulator In the simulator class (derived from `SimCase` a line must be added to the `scan()` routine. In the same place that you do

```
lineq->attachCommAdm(*gp_adm);
```

you must also do

```
parms->attachCommAdm(*gp_adm);
```

Now all the pARMS solvers and preconditioners will be available from the menusystem. To see the list of solvers, run the application interactively, and type:

```
sub LinEqAdmFE
sub LinEqSolver_prm
help basic method
```

Notice `valid answer`, it should look like this:

```
String, alternatives: /GaussElim/Jacobi/SOR/SSOR/ConjGrad/Symmlq/\
CGS/BiCGStab/TFQMR/MinRes/GMRES/Orthomin/AMG/SymMinRes/pARMS_fgmresd/\
pARMS_dgmresd/pARMS_bcgstabd/
```

To see the list of preconditioners, run the application interactively, and type:

```
sub LinEqAdmFE
sub Precond_prm
help preconditioning type
```

Notice `valid answer`, it should look like this:

```
String, alternatives: /PrecNone/PrecUserDefLU/PrecUserDefInv/\
PrecUserDefMat/PrecUserDefProc/PrecRILU/PrecJacobi/PrecSOR/\
PrecSSOR/PrecJacobiIter/PrecSORIter/PrecSSORIter/PrecPARMSAddIlu0/\
PrecPARMSAddIlut/PrecPARMSAddIluk/PrecPARMSAddArms/\
PrecPARMSLschIlu0/PrecPARMSLschIlut/PrecPARMSLschIluk/\
PrecPARMSLschArms/PrecPARMSRschIlu0/PrecPARMSRschIlut/\
PrecPARMSRschIluk/\PrecPARMSRschArms/PrecPARMSSchGilu0/\
PrecPARMSSchSgs/
```

References

[1] IEEE standard for binary floating-point arithmetic (IEEE 754). Technical report, The Institute of Electrical and Electronics Engineers, 1985.

[2] Nelson H. F. Beebe. Using C and C++ with Fortran. `http://www.math.utah.edu/software/c-with-fortran.html#data-type-diffs`, 2001.

[3] Hyoung Joong Kim; Hyung Soo Kim; Kyung Choi; Hyang beom Lee; Hyun Kyo Jung; Song-yop Hahn. Cost-effective parallel preconditioner for network-based computing. *Magnetics, IEEE Transactions on*, 1997.

[4] Are Magnus Bruaset. *A survey of preconditioned iterative methods*. Longman Scientific & Technical, 1995.

[5] X. Cai, E. Acklam, H. P. Langtangen, and A. Tveito. *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Springer-Verlag, 2003.

[6] Free Software Foundation Inc. Gnu lesser general public license, version 2.1. `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`.

[7] inuTech GmbH. Diffpack documentation for gmres. `http://www.diffpack.com/diffpack/refmanuals/current/classGMRES.html`. As found 2007-10-22 14:34.

[8] inuTech GmbH. Diffpack: Software for finite element analysis and partial differential equations. `http://diffpack.com`.

[9] Danny Kalev. Integrating C and C++. `http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=180&rl=1`, 2004.

[10] Hans Petter Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming. Second Edition, Volume 1 in Textbooks in Computational Science and Engineering*. Springer, 2003.

[11] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations. Documentation as shipped with SPARSKIT, 1994.

[12] Yousef Saad. parms home page. `http://www-users.cs.umn.edu/ saad/software/pARMS/index.html`.

[13] Yousef Saad. Sparskit home page. `http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html`.

[14] Marc Snir and William Gropp. *MPI: The Complete Reference*. The MIT Press, 1998.

[15] Martin Burheim Tingstad. Improving inter-subdomain communication and load-balancing for the parallel diffpack library. Master's thesis, Universitetet i Oslo, 2007.

[16] Wikipedia. Name mangling in Fortran. http://en.wikipedia.org/wiki/Name_mangling#Name_mangling_in_Fortran. As found 2007-01-15 17:56.

[17] Masha Sosonkina Yousef Saad. parms: A package for solving general sparse linear systems on parallel computers. In *Proceedings of the 4th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, pages 446 – 457, 2001.

[18] Masha Sosonkina Yousef Saad. parms: A package for the parallel iterative solution of general large sparse linear systems, users guide. Technical report, University of Minnesota, 2003.